Extrinsically Typed Operational Semantics for Functional Languages

Matteo Cimini
University of Massachusetts Lowell
USA
matteo_cimini@uml.edu

Dale Miller
Inria & LIX, École Polytechnique
France
dale.miller@inria.fr

Jeremy G. Siek
Indiana University
USA
jsiek@indiana.edu

Abstract

We present a type system over language definitions that classifies parts of the operational semantics of a language in input, and models a common language design organization. The resulting typing discipline guarantees that the language at hand is automatically type sound.

Thanks to the use of types to model language design, our type checker has a high-level view on the language being analyzed and can report messages using the same jargon of language designers.

We have implemented our type system in the LANG-N-CHECK tool, and we have applied it to derive the type soundness of several functional languages, including those with recursive types, polymorphism, exceptions, lists, sums, and several common types and operators.

CCS Concepts: • **Software and its engineering** → *Formal language definitions*; • **Theory of computation** → *Type theory*.

Keywords: Extrinsic type systems, type soundness, functional languages

ACM Reference Format:
Matteo Cimini, Dale Miller, and Jeremy G. Siek. 2020. Extrinsically Typed Operational Semantics for Functional Languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, Article 1, 18 pages.

1 Introduction

Type systems play a major role in *program analysis*. Can they play a similar role in *language analysis*, i.e. the analysis of the meta-theoretic properties of programming languages?

The research line on intrinsic typing has long investigated this question, and has demonstrated that type systems can be an effective tool for the analysis of languages. In this approach, a language is implemented within a host type theory with strong meta-theoretic properties [Church 1940; Harper and Stone 2000; Poulsen et al. 2018; Rouvoet et al. 2020]. If the implementation type checks then the language is type sound. For example, if we can make a language definition type check within a type theory with exhaustive pattern-matching that is strongly normalizing then the progress theorem automatically holds.

In this paper, we take a different perspective in the type checking of languages for their soundness: we intend to use types and type systems in the same way they are used in program analysis [Cardelli 2004; Pierce 2002], i.e., to model a high-level organization and discipline to ensure that some property holds. The most common way to perform type checking is with an external function that analyzes programs, that is, an *extrinsic* type system. Our goal is to present an extrinsic type system that accepts or rejects a programming language definition (syntax, type system, operational semantics) and guarantees that accepted languages are type sound.

A Meta Type System for Type Soundness. To avoid naming confusion, we use the term *meta type system* for a type system that analyzes a language definition rather than a program[1]. Our main contribution is a meta type system that ensures that the various parts of a language definition (value declarations, typing rules, reduction rules, evaluation contexts, and so on) are all in order so that type soundness automatically holds. Our meta type system models a high-level organization and discipline that are not novel. Conversely, they capture invariants that language designers have been using for a long time, and makes them explicit and formally defined. For example, our meta type system traverses the grammar and typing rules of the language at hand in order to apply the common classification of operators as *introduction forms* (such as $\lambda x : T.e$ in the simply typed λ-calculus (STLC)), *elimination forms* (such as application), *derived operators* (such as `let` and `letrec`), *errors* and *error handlers* (such as `try`). This classification paves the way for a high-level analysis of a language definition. For example, our meta type system can check the β-rule of STLC $(\lambda x : T.e)\ v \longrightarrow e[v/x]$ in the following way. (Below, the Classification meta-variable contains information about the role of operators and characterizes values. The application operator is named as `app`, `funType` is the function type, and

[1] The word *extrinsic* can be confusing when we dive in the technical part of the paper, as our meta type system needs to analyze a type system which is itself extrinsic. We therefore say *meta type system*.

SLE'20, November, 2020, Online
2020.

Section 3.2 presents complete details.)

$$\frac{\begin{array}{c}\text{Classification} \vdash \texttt{app} : \text{elim } \texttt{funType} \\ \text{Classification} \vdash (\lambda x : T.\ e) : \text{value } \texttt{funType} \\ \text{Contexts} \vdash \texttt{app} : \{1, 2\}\end{array}}{\begin{array}{l}\text{Contexts} \\ \mid \text{Classification}\end{array} \vdash (\texttt{app}\ (\lambda x : T.\ e)\ v) \longrightarrow e[v/x]}$$

This is a meta-level typing rule that type checks a reduction rule. Here, the first line of premises says that the type of the operator app is "elim funType", which means that the application is an elimination form of the function type, previously classified as such in Classification by other parts of the meta type system. It then requires that the argument being eliminated, which is highlighted and is sometimes called the *principal argument* [Harper 2012], be a value of the function type, which we check by making sure that the abstraction is of type "value funType". This materializes a common design in programming languages in which *elimination forms manipulate values of some data type*. As exemplified above, it is typical to apply this approach by assigning a principal argument to an elimination form of a type and then by defining its reduction rules by case analysis on the values of that type.

The meta-level typing rule above also checks that the arguments at positions 1 and 2 must be evaluation contexts for the application, as those arguments need to be evaluated for the β-rule to apply. Contexts $\vdash$ app : $\{1, 2\}$ enforces that the definition of evaluation contexts for application have E in both the first and second sub-expression positions, as in the grammar: $E ::= (\texttt{app}\ E\ e) \mid (\texttt{app}\ v\ E) \mid \cdots$.

Analogously, if the language at hand had lists and the reduction rule $\texttt{head}\ (\texttt{cons}\ v_1\ v_2) \longrightarrow v_1$, our meta type system would use an instance of the rule above and detect that head is an elimination form for lists. It would then check that $(\texttt{cons}\ v_1\ v_2)$ is a value for lists, and that the evaluation contexts are defined appropriately.

Our full meta type system captures design principles that are based on the above as well as other common language design invariants. It applies to functional languages with a typing relation of the form $\Gamma \vdash e : T$ and a reduction relation of the form $e \longrightarrow e$ based on small-step semantics and evaluation contexts. Ultimately, we have proved that languages that conform to our meta type system are type sound.

***Implementation:* LANG-N-CHECK.** We have implemented our meta type system in the LANG-N-CHECK tool [Cimini 2015]. The tool works with language definitions such as that in Figure 1, which contains the operational semantics of System F [Girard 1972; Reynolds 1974] with booleans. Language definitions use a domain-specific language that is close to pen&paper definitions in textbooks [Harper 2012; Pierce 2002] and research papers. The reason for this is that we wish to use the tool in courses on programming languages.

```
1  Expression E ::= x | (abs T (x)E) | (absT (X)E) | (app E E)
2                   | (appT E T) | (tt) | (ff) | (if E E E)
3  Type T ::= (bool) | (arrow T T) | (all (X)T)
4  Value V ::= (abs T (x)E) | (absT (X)E) | (tt) | (ff)
5  Error ::=
6  Context C ::= [] | (app C e) | (app v C) | (appT C T)
7                | (if C e e)
8
9  Gamma |- (abs T1 E) : (arrow T1 T2) <==
10                              Gamma, x : T1 |- E : T2.
11 Gamma |- (absT E) : (all T) <== Gamma, X |- E : T.
12 Gamma |- (app E1 E2) : T2 <== Gamma |- E1 : (arrow T1 T2)
13                              /\ Gamma |- E2 : T1.
14 Gamma |- (appT E T1) : T2[T1/X] <== Gamma |- E : (all T2).
15 Gamma |- (tt) : (bool).
16 Gamma |- (ff) : (bool).
17 Gamma |- (if E1 E2 E3) : T <== Gamma |- E1 : (bool)
18                              /\ Gamma |- E2 : T
19                              /\ Gamma |- E3 : T.
20
21 (app (abs T E) V) --> E[V/x].
22 (appT (absT E) T) --> E[T/X].
23 (if (tt) E1 E2) --> E1.
24 (if (ff) E1 E2) --> E2.
```

Figure 1. Example input of LANG-N-CHECK: language definition of System F with booleans. We do not mention binding variable in lines such as 21 and 22 because x is the bound expression variable by default in the tool, and X is the type variable by default.

Language definitions that type check are type sound. *What happens if our meta type system fails?* One of the differences between our approach and intrinsic typing is that since we directly model language design with types then the meta type checker has a high-level view on the language being analyzed and can report messages using the same jargon of language designers. For example, if we forgot the context (if C e e) at line 7, LANG-N-CHECK would reject the specification with the error message *"The principal argument of the elimination form* if *is not declared as evaluation context, hence some programs may get stuck"*. We are not aware of intrinsic type systems for languages whose type error messages refer to terms such as "principal argument" and "elimination form".

We have applied LANG-N-CHECK to a plethora of functional languages and checked their type soundness, including the simply typed λ-calculus and its variants with integers, booleans, pairs, lists, sums, tuples, fix, let, letrec, unit, universal types, recursive types, option types, exceptions, list operations such as append, map, mapi (also depends on the position in the list of the current element being processed), filter, filteri, range, list length, reverse, and the recursor of natural numbers (natrec). We have also considered different evaluation strategies for the features mentioned above: call-by-value, call-by-name and a parallel reduction strategy (both function and argument can evaluate non-deterministically), as well as lazy pairs, lazy lists, and lazy tuples, and both left-to-right and right-to-left evaluations. LANG-N-CHECK compiles the languages that type check successfully into the Abella theorem prover [Baelde et al. 2014] and automatically generates their machine-checked proof of type soundness.

This gives us high confidence that, besides the theoretical guarantees of our paper, also our implementation is reliable.

Closely Related Work. Cimini reports that LANG-N-CHECK has been used to teach programming languages theory [Cimini 2019]. We would like to clearly distinguish that work from this paper. That work briefly (and incompletely) describes the capabilities of LANG-N-CHECK[2], makes the point that error messages are informative, sets forth the thesis that a tool such as LANG-N-CHECK can be beneficial in a teaching context, and describes the studies that have been made, and that are to be made in the future. This paper, instead, provides a formal meta type system, gives a complete description of the organization principles that guide the meta type system, provides a proof of correctness, and reports on the many languages that have been checked as type sound using LANG-N-CHECK.

Intrinsic vs Extrinsic Typing? So Far, Intrinsic Wins. We have described the difference between intrinsic and extrinsic typing above, and also pointed out the benefits of the latter. Where does extrinsic typing of languages stand as compared to intrinsic typing?

A recent result by Poulsen et al has demonstrated that intrinsic typing can be applied to practical languages, and in particular to a large subset of Middleweight Java [Poulsen et al. 2018]. Our work is limited to the functional realm, and we acknowledge that the state of the art of intrinsic typing is substantially ahead.

The research line of intrinsic typing can be seen as occurring in two stages. **Stage 1** laid down the idea and the practice of intrinsic typing. **Stage 2** demonstrated that intrinsic typing can scale, as was recently shown by Poulsen et al 2018.

Research in extrinsic typing of languages may need to follow a similar path. We believe that the research in this paper helps to establish **Stage 1**. It has been challenging to reach **Stage 2** for intrinsic typing: we leave scaling up our approach for future work.

Summary of our Contributions. This paper makes the following contributions.

1. We devise a new approach to language analysis based on extrinsic typing of operational semantics. Our meta type system models a common high-level organization and discipline on language definitions that guarantee their type soundness (Section 3).
2. We prove that language definitions accepted by our meta type system are type sound (Section 4). In some sense, this result proves that the invariants that language designers have been using for years are correct at some general scale.
3. We implement the meta type system in LANG-N-CHECK (Section 5). We have applied the tool to derive the type soundness of several functional languages, including those with recursive types, polymorphism, exceptions, lists, sums, and several common types and operators.

[2]That paper refers to a tool with another name because LANG-N-CHECK is used within a larger tool.

2 Language Definitions

We briefly review how languages are defined in small-step operational semantics. This section serves to fix the terminology that we use in the rest of the paper.

Figure 2 shows the definition of F_{exc}, which essentially is System F extended with exceptions. The language defines a series of syntactic categories defined by a BNF grammar. The syntactic categories Type and Expression define the types and the expressions of the language. Next, language designers decide which expressions constitute *values*. These are the results of successful computations. Values are defined with the syntactic category Value. Similarly, language designers designate which expression constitute the *error*, which is the outcome when computations fail. The error is defined with the syntactic category Error. Context defines the *evaluation contexts*, which prescribe within what contexts reduction can take place. Similarly, Error Context defines the *error contexts*, which are those contexts in which we are allowed to detect that an error has occurred. (They typically differ from evaluation contexts when the language includes error handlers.)

Next, F_{exc} defines its type system with a relation of the form $\Gamma \vdash e : T$, which is inductively defined with a set of inference rules. The dynamic semantics here is of the form $e \longrightarrow e$ and is also inductively defined by a set of inference rules called reduction rules.

In the setting of small-step operational semantics and languages with errors, the following is the statement of type soundness. ($\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$).

Definition 2.1 (Type Soundness)**.** A language $\mathcal{L}$ *is type sound* whenever for all expressions e, e', and types T, if $\emptyset \vdash e : T$ and $e \longrightarrow^* e'$ then either

- e' is a value such that $\emptyset \vdash e' : T$,
- e' is an error, or
- there exists e'' such that $e' \longrightarrow e''$ and $\emptyset \vdash e'' : T$.

3 A Meta Type System for Type Soundness

We work with a formal representation for language definitions. We define a *language definition* $\mathcal{L}$ as an 8-tuple

(Type, Expression, Value, Error, Context, Error Context, Type System, Dynamic Semantics).

$$
\begin{array}{lllll}
\text{Type} & T & ::= & \top \mid T \to T \mid \forall X.T \\
\text{Expression} & e & ::= & x \mid \lambda x : T.e \mid e\ e \mid \Lambda X.e \mid e\ [T] \\
 & & & \mid \texttt{raise}\ e \mid \texttt{try}\ e\ \texttt{with}\ e \\
\text{Value} & v & ::= & \lambda x : T.e \mid \Lambda X.e \\
\text{Error} & er & ::= & \texttt{raise}\ v \\
\text{Context} & E & ::= & E\ e \mid v\ E \mid E\ [T] \mid \texttt{raise}\ E \mid \texttt{try}\ E\ \texttt{with}\ e \\
\text{Error Ctx} & F & ::= & F\ e \mid v\ F \mid F\ [T] \mid \texttt{raise}\ F
\end{array}
$$

Type System $\boxed{\Gamma \vdash e : T}$

$$\text{(T-VAR)}\ \ \Gamma, x : T \vdash x : T$$

$$\text{(T-ABS)}\ \ \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T.e : T_1 \to T_2}$$

$$\text{(T-APP)}\ \ \frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$$

$$\text{(T-TABS)}\ \ \frac{\Gamma, X \vdash\ e : T}{\Gamma \vdash\ \Lambda X.e : \forall X.T}$$

$$\text{(T-TAPP)}\ \ \frac{\Gamma \vdash\ e : \forall X.T_2}{\Gamma \vdash\ (e\ [T_1]) : T_2[T_1/X]}$$

$$\text{(T-RAISE)}\ \ \frac{\Gamma \vdash e : \top}{\Gamma \vdash \texttt{raise}\ e : T}$$

$$\text{(T-TRY)}\ \ \frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : \top \to T}{\Gamma \vdash\ \texttt{try}\ e_1\ \texttt{with}\ e_2 : T}$$

Dynamic Semantics $\boxed{e \longrightarrow e}$

$$(\lambda x : T.e)\ v \longrightarrow e[v/x]\ \ \text{(BETA)} \qquad \Lambda X.e\ [T] \longrightarrow e[T/X]\ \ \text{(R-TAPPT)}$$

$$\texttt{try}\ v\ \texttt{with}\ e \longrightarrow v\ \ \text{(R-TRY-SUCCESS)} \qquad \texttt{try}\ (\texttt{raise}\ v)\ \texttt{with}\ e \longrightarrow (e\ v)\ \ \text{(R-TRY-RAISE)}$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}\ \text{(CTX)} \qquad F[er] \longrightarrow er\ \text{(ERR-CTX)}$$

Figure 2. Type system and dynamic semantics of F_{exc}.

where we place the components of Fig 2 in the tuple $\mathcal{L}$. To make an example, F_{exc} is represented as

(Types = $T ::= \top \mid T \to T \mid \forall X.T$,
Expression = $e ::= x \mid \lambda x : T.e \mid e\ e \mid \Lambda X.e \mid e\ [T] \mid \texttt{raise}\ e \mid \texttt{try}\ e\ \texttt{with}\ e$,
Value = $v ::= \lambda x : T.e \mid \Lambda X.e$,
Error = $er ::= \texttt{raise}\ v$,
Context = $E ::= E\ e \mid v\ E \mid E\ [T] \mid \texttt{raise}\ E \mid \texttt{try}\ E\ \texttt{with}\ e$,
Error Context = $F ::= F\ e \mid v\ F \mid F\ [T] \mid \texttt{raise}\ F$,
Type System = {(T-VAR), (T-ABS), (T-APP), (T-TABS), (T-TAPP), (T-RAISE), (T-TRY)},
Dynamic Semantics = {(BETA), (R-TAPP), (R-TRY-SUCCESS), (R-TRY-RAISE), (R-CTX), (R-ERRCTX)})

where we simply wrote the name of rules, although the tuple contains the actual inference rules.

In this section we define a meta type system with judgment $\vdash \mathcal{L}$ that makes sure that soundness automatically holds.

Our setting must accommodate syntax with some generality. Therefore, we assume that operators are defined in abstract syntax rather than concrete syntax. For example, we have `if` $e\ e\ e$ rather than `if` e `then` e `else` e. We use xs as variables for expressions and (capitalized) Xs for types. Without loss of generality, we assume that the type annotations of an operator all appear first, followed by types with a bound X, followed by expressions with a bound x, followed by expressions with a bound X, and finally followed by (not bound) expression arguments. This leads to a general shape that is similar to Harper's abstract binding trees [Harper 2012] and is $op\ \ \overline{T}\ \ \overline{(X).T}\ \ \overline{(x).e}\ \ \overline{(X).e}\ \ \overline{e}$, where op is an operator, and $\overline{\cdot}$ denotes sequences. To make an example, abstraction fits that general shape as (`abs` T $(x).e$) and a type-annotated `cons` fits that shape as (`cons` $T\ e\ e$).

Our meta type system is an inference system that type checks parts of languages, including inference rules. To avoid confusion, we write the parts of the language being type checked in blue color. To help our presentation, we some times make examples with operators that are not in F_{exc}.

Figure 3 contains the main judgement for $\vdash \mathcal{L}$, handled by rule [MAIN]. This rule simply checks all the components of the language in the way described in the following sections.

3.1 Meta Type Systems for the Syntactic Categories

The meta type system for type checking the syntactic categories are in Figure 3, as well. The language design invariants that are at play at this point are:

- **Val:** *Any argument that is required to be evaluated to a value for a definition to apply must be declared as evaluation context.* For example, ***Val*** enforces that if (`cons` $v\ v$) is declared as value then Context must contain productions of the form (`cons` E _) and (`cons` _ E), where _ can be e or v in both occurrences. Indeed, if one of these were missing, say the first, the expression (`cons` ($(\lambda x.x)$ `nil`) `nil`) would be stuck. This expression is not a value and does not take any step. Type soundness would then be jeopardized.
- **Ctx:** *Evaluation contexts have no circular dependencies w.r.t. the evaluation of their arguments.* For example, we must forbid context declarations such as (`cons` $E\ v$) and (`cons` $v\ E$). Such declarations make the expression (`cons` ($(\lambda x.x)$ `nil`) ($(\lambda x.x)$ `nil`)) stuck because the first argument does not start evaluating until the second is a value, while the second waits for the first to become a value, which won't start evaluating.
- **ErrCtx:** *Error contexts are evaluation contexts minus the error handler at the principal argument (where the error is expected to appear to be handled).* This says that the context `try` F `with` e should not be an error context. Indeed, (ERR-CTX) should not apply to derive the reduction `try` (`raise` v) `with` $e \longrightarrow$ `raise` v, as we expect the semantics of `try` to handle the error.

Type Checking **Type.** The judgement $\vdash$ Type in Fig. 3 checks some restrictions on the grammar for types. In particular, we allow types to be a constant c applied to Ts, or applied to Ts with bound variables Xs (for types quantified by types). On one hand, this accommodates universal types

and recursive types but also prevents expressions from being used in types, ruling out dependent types and refinement types. These latter features are rather advanced and we leave them for future work.

Type Checking Expression**.** We use exp^- to denote an expression derived by the grammar for Expression. The judgement $\vdash$ Expression checks that each grammar production is of the form $op\ \overline{T}\ \overline{(X).T}\ \overline{(x).e}\ \overline{(X).e}\ \overline{e}$, which we have discussed previously. Notice that no other syntactic categories are allowed so far. This means that we rule out layered grammars such as expressions vs statements, which we plan to address in the future.

Type Checking Value**.** The judgement Context $\vdash$ Value checks the value declarations. We refer to exp for exp^- in which we can use v. There are two aspects that are checked for values. The first is that they make use of ev, which denotes a variable that can be either e or v. The second aspect is that we check the language design invariant ***Val***, defined above. This check is done with Context $\vdash$ op : *contextPositions*, which informs about the set of argument positions that are declared as evaluation contexts for op. For example, if Context has declarations $E ::= (\texttt{cons}\ E\ e) \mid (\texttt{cons}\ v\ E)$, we have Context $\vdash$ `cons` : $\{1, 2\}$. Notice that E cannot appear underneath a binder, so we forbid evaluation underneath binders. Once we have computed the set *contextPositions*, we then check that the argument positions in a value declaration that are vs, i.e., that are required to be values, are in that set. To make an example, the value declaration $v ::= (\texttt{cons}\ v\ v)$ is well-typed because the vs appear in argument positions 1 and 2, which are in $\{1, 2\}$, contextual positions for `cons`.

Type Checking Error**.** The judgement Context $\vdash$ Error checks the error declaration, if present. The invariant ***Val*** is checked here, as well. For example, `raise` E must be an evaluation context to prevent expressions such as `raise` $(\lambda x : T.e\ \texttt{nil})$ from being stuck and thereby jeopardizing type soundness.

Type Checking Context**.** We use evE to denote a variable that can be e, v or E, and we refer to ctx as those expressions where Es and vs can appear. For example, $(\texttt{cons}\ E\ v)$ is a valid ctx. The judgement Context $\vdash$ Context checks that each context declaration has exactly one occurrence of E. That is, $(\texttt{cons}\ E\ E)$ is not a valid ctx. (We enforce this with the existential quantification with uniqueness $\exists!$.) It also enforces language design invariants **Val** and **Ctx**. The latter is ensured with the use of the function `acyclic` which performs this check through a graph representation of the dependencies at play. To be precise, we have an edge for each context declaration from the index position of E to the index positions of vs. Context declarations $(\texttt{cons}\ E\ e)$ and $(\texttt{cons}\ v\ E)$, i.e., left-to-right evaluation, induce the graph $\{2 \mapsto 1\}$, which is acyclic. The bad context declarations $(\texttt{cons}\ E\ v)$ and $(\texttt{cons}\ v\ E)$, instead, induce the graph $\{1 \mapsto 2, 2 \mapsto 1\}$, which contains a cycle and is rejected. The function `acyclic` performs a standard topological sort and so we omit its definition.

The last syntactic category that must be checked is Error Context. However, in order to check invariant **ErrCtx** we need to know whether an error handler exists. We therefore postpone this check until after we present the meta type system for Type System.

3.2 Meta Type System for Typing Rules

Figure 4 contains the meta type system for checking the typing rules of the language. The typing rules classify the operators of the language w.r.t. their role. Operators are classified as *introduction forms*, *elimination forms*, *derived operators*, *the error* and *error handlers*.

[T-MAIN] (Figure 4) is the main rule. It checks all the typing rules and returns the complete classification of the operators in the structure Classification, which we explain. When checking each typing rule, we return a *role map* B which can be of three possible forms: 1) $op \mapsto R$, where op is an operator and R is the role that this operator plays in the language, 2) $exp \mapsto$ value c that denotes that exp is a value for the type constructor c, or 3) $exp \mapsto$ error, which means that exp is an error.

The meta type system collects all the role maps $B_1, \ldots, B_n$, for which we check $\texttt{compatible}(B_1, \ldots, B_n)$. This check ensures that each operator has only one role. For example, `cons` cannot be both a value and an elimination form. Ultimately, the collection of all the role maps forms Classification.

Anatomy of a Typing Rule. Below we discuss how the meta type system checks each typing rule. Before proceeding, we take a closer look at the typing rule for `let` and type abstraction in system F.

$$\text{(T-LET)}\quad \frac{\Gamma, \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : T_2} \qquad \text{(T-TABS)}\quad \frac{\Gamma, X \vdash e : T}{\Gamma \vdash \Lambda X.e : \forall X.T}$$

We say that `let` is the *subject* of the typing rule (T-LET), and T_2 is its *assigned type*. We impose a common form for type environments: they can be used as $\Gamma \vdash e : T$ and Γ can have bindings $x : T$ (as in (T-LET)), and/or type variables X (as in (T-ABST)). We also impose that typing rules type check *all* their e arguments, as done above and in virtually any typing rule we know.

[T-VALUE] detects the introduction forms following the common design principle that *introduction forms build values of some data type*. To this aim, [T-VALUE] imposes that the assigned type of the typing rule has the form $(c\ \overline{T})$, as highlighted, that is, a type constructor applied to arguments. [T-VALUE] also checks that op forms a value. To make some examples, below [T-VALUE] is instantiated to check (T-ABS)

$$\boxed{\vdash \mathcal{L}}$$

$$\frac{\vdash \text{Type} \quad \vdash \text{Expression} \quad \text{Context} \vdash \text{Value} \quad \text{Context} \vdash \text{Error} \quad \text{Context} \vdash \text{Context}}{\begin{array}{c}\text{Value} \mid \text{Error} \mid \text{Dynamic Semantics} \vdash \text{Type System} : \text{Classification} \\ \text{Classification} \mid \text{Context} \vdash \text{Error Context} \\ \text{Context} \mid \text{Classification} \mid \text{Type System} \vdash \text{Dynamic Semantics}\end{array}}{\vdash (\text{Types, Expression, Value, Error, Context, Error Context, Type System, Dynamic Semantics})}$$

$$\boxed{\vdash \text{Type and} \vdash ty}$$

$$\frac{\vdash ty_1 \quad \ldots \quad \vdash ty_n}{\vdash T ::= ty_1 \mid \ldots \mid ty_n} \qquad \frac{}{\vdash c\ \overline{T}\ \overline{(X).T}}$$

$$\boxed{\vdash \text{Expression and} \vdash exp^-}$$

$$\frac{\vdash exp_1^- \quad \ldots \quad \vdash exp_n^-}{\vdash e ::= exp_1^- \mid \ldots \mid exp_n^-} \qquad \frac{}{\vdash op\ \overline{T}\ \overline{(X).T}\ \overline{(x).e}\ \overline{(X).e}\ \overline{e}}$$

$$\boxed{\text{Context} \vdash \text{Value and Context} \vdash exp}$$

$$\frac{\text{Context} \vdash exp_1 \quad \ldots \quad \text{Context} \vdash exp_n}{\text{Context} \vdash v ::= exp_1 \mid \ldots \mid exp_n} \qquad \frac{\begin{array}{c}\text{Context} \vdash op : contextPositions \\ \forall i \in \{i \mid ev_i = v\}, i \in contextPositions\end{array}}{\text{Context} \vdash op\ \overline{T}\ \overline{(X).T}\ \overline{(x).e}\ \overline{(X).e}\ \overline{ev}}$$

$$\boxed{\text{Context} \vdash \text{Error and Context} \vdash op : contextPositions}$$

$$\frac{}{\text{Context} \vdash er ::= \epsilon} \qquad \frac{\text{Context} \vdash exp}{\text{Context} \vdash er ::= exp} \qquad \frac{contextPositions = \{i \mid op\ \overline{args} \in \text{Context and } E \in \overline{args} \text{ at position } i\}}{\text{Context} \vdash op : contextPositions}$$

$$\boxed{\text{Context} \vdash \text{Context and Context} \vdash ctx}$$

$$\frac{\text{Context} \vdash ctx_1 \quad \ldots \quad \text{Context} \vdash ctx_n \quad \texttt{acyclic(Context)}}{\text{Context} \vdash E ::= ctx_1 \mid \ldots \mid ctx_n} \qquad \frac{\begin{array}{c}\exists!(evE_i = E) \\ \text{Context} \vdash op : contextPositions \\ \forall i \in \{i \mid ev_i = v\}, i \in contextPositions\end{array}}{\text{Context} \vdash op\ \overline{T}\ \overline{(X).T}\ \overline{(x).e}\ \overline{(X).e}\ \overline{evE}}$$

Figure 3. Meta Type system: Main Judgement and Syntactic Categories, with the exception of Error Context, which is defined at the end of Section 3.2. ty is an expression derived by the grammar for Type. exp is exp^- in which we can use v variables. ctx is derived by the grammar for Context. exp^- is derived by the grammar for Expression. ev, is a variable that can be either e or v. evE, is a variable that can be either e or v or E.

and the typing rule for the list operator `cons`.

$$\begin{array}{l} v ::= \lambda x : T.e \ \cdots \\ \mid \text{Error} \\ \mid \text{Dynamic Sem.} \end{array} \quad \vdash \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T_1.e : T_1 \to T_2} : \begin{array}{c} \lambda x : T.e \\ \mapsto \\ \text{value} \to \end{array}$$

$$\begin{array}{l} v ::= \texttt{cons}\ v\ v \ \cdots \\ \mid \text{Error} \\ \mid \text{Dyn. Sem.} \end{array} \quad \vdash \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : (\texttt{List}\ T)}{\Gamma \vdash (\texttt{cons}\ e_1\ e_2) : \texttt{List}\ T} : \begin{array}{c} \texttt{cons}\ v\ v \\ \mapsto \\ \text{value}\ \texttt{List} \end{array}$$

[T-VALUE] helps satisfy the progress theorem because when the values of the language are type checked with this rule it is immediate to prove the lemmas for the canonical forms of the language.

[T-ELIM] classifies elimination forms following a common pattern:

Elim: *elimination forms manipulate/inspect values of some data type at their principal argument.*

Accordingly, [T-ELIM] imposes that the typing rule assigns a complex type $(c\ \overline{T})$ to the principal argument of op_1. To simplify our presentation, we assume without loss of generality that the principal argument is always the first expression variable (e_1). We expect that the behavior of the operator is defined by case analysis on some values. Therefore, [T-ELIM] expects to find at least a reduction rule of the form $(op_1\ \boxed{(op_2\ \overline{args_2})}\ \overline{args_1}) \longrightarrow exp^+$, that is, the principal argument is a complex expression, as highlighted. Reduction rules typically follow this pattern, examples are

Type Environment	Γ^*	::=	$\Gamma \mid \Gamma, x : T \mid \Gamma, X$
Role Maps	B	::=	$op \mapsto R \mid exp \mapsto \text{value } c \mid exp \mapsto \text{error}$
Roles	R	::=	$\text{elim } c \mid \text{derived} \mid \text{errorHandler}$

$$\boxed{\text{Value} \mid \text{Error} \mid \text{Dynamic Semantics} \vdash \text{Type System} : \text{Classification}}$$

[T-MAIN]

$$\frac{\begin{array}{l}\text{Values} \mid \text{Error} \mid \text{Dynamic Semantics} \vdash \phi_1 : B_1 \\ \qquad\qquad \ldots \\ \text{Values} \mid \text{Error} \mid \text{Dynamic Semantics} \vdash \phi_n : B_n \\ \qquad\qquad \texttt{compatible}(B_1, \ldots, B_n)\end{array}}{\text{Values} \mid \text{Error} \mid \text{Dynamic Semantics} \vdash \{\phi_1, \ldots, \phi_n\} : \{B_1, \ldots, B_n\}}$$

[T-VALUE]

$$\frac{(op\ \overline{args}) \in \text{Values}}{\text{Values} \mid \text{Error} \mid \text{Dynamic Semantics} \vdash \dfrac{\Gamma^* \vdash e_1 : ty_1 \quad \ldots \quad \Gamma^* \vdash e_n : ty_n}{\Gamma \vdash (op\ \overline{T'}\ e_1\ \cdots\ e_n) : (c\ \overline{T})} : (op\ \overline{args}) \mapsto \text{value } c}$$

[T-ELIM]

$$\frac{(op_1\ (op_2\ \overline{args_2})\ \overline{args_1}) \longrightarrow exp^+ \in \text{Dynamic Semantics} \qquad op_2 \in \text{Values}}{\text{Values} \mid \text{Error} \mid \text{Dynamic Semantics} \vdash \dfrac{\Gamma \vdash e_1 : (c\ \overline{T}) \quad \ldots \quad \Gamma^* \vdash e_n : ty_n}{\Gamma \vdash (op_1\ \overline{T'}\ e_1\ \cdots\ e_n) : ty} : op_1 \mapsto \text{elim } c}$$

[T-ERRHANDLER]

$$\frac{(op_1\ (op_2\ \overline{args_2})\ \overline{args_1}) \longrightarrow exp^+ \in \text{Dynamic Semantics} \qquad op_2 \in \text{Error}}{\text{Values} \mid \text{Error} \mid \text{Dynamic Semantics} \vdash \dfrac{\Gamma^* \vdash e_1 : ty_1 \quad \ldots \quad \Gamma^* \vdash e_n : ty_n}{\Gamma \vdash (op_1\ \overline{T}\ e_1\ \cdots\ e_n) : ty} : op_1 \mapsto \text{errorHandler}}$$

[T-DERIVED]

$$\frac{\forall\ (op\ \overline{terms}) \longrightarrow exp^+ \in \text{Dynamic Semantics},\ \overline{terms} \text{ contains only variables}}{\text{Values} \mid \text{Error} \mid \text{Dynamic Semantics} \vdash \dfrac{\Gamma^* \vdash e_1 : ty_1 \quad \ldots \quad \Gamma^* \vdash e_n : ty_n}{\Gamma \vdash (op\ \overline{T}\ e_1\ \cdots\ e_n) : ty} : op \mapsto \text{derived}}$$

[T-ERROR]

$$\frac{(op\ \overline{args}) \in \text{Error} \qquad T \notin (\overline{T'} \cup \texttt{vars}(ty_1) \cup \ldots \cup \texttt{vars}(ty_n))}{\text{Values} \mid \text{Error} \mid \text{Dynamic Semantics} \vdash \dfrac{\Gamma^* \vdash e_1 : ty_1 \quad \ldots \quad \Gamma^* \vdash e_n : ty_n}{\Gamma \vdash (op\ \overline{T'}\ e_1\ \cdots\ e_n) : T} : (op\ \overline{args}) \mapsto \text{error}}$$

Figure 4. Meta Type system: Typing Rules. For simplicity, each e_i in $e_1, \ldots, e_n$ may be an expression variable or a bound expression such as $(x).e$. The sequences $\overline{T}$ contain types, each of which may also be a bound type such as $(X).T$. Sequences $\overline{args}$ contain variables, which may be under a binder as well. Also, sequences in this figure have distinct variables.

$(\texttt{app}\ (\lambda\ x.e)\ e) \longrightarrow e[v/x]$ for the elimination form `app`, and $(\texttt{head}\ (\texttt{cons}\ v_1\ v_2)) \longrightarrow v_1$ for the elimination form `head`. Here, the left-hand side of reduction rules have an expression exp^+, which denotes an expression exp in which substitutions of the form $exp[exp/x]$ and $exp[ty/x]$ can occur.

Some of the reduction rules of error handlers have the same pattern of that for elimination forms, but an error appears as principal argument. Therefore, to distinguish whether the operator is an elimination form rather than an error handler, we check that op_2 is a value.

To make some examples, below we instantiate [T-ELIM] for checking (T-APP), and the typing rule for `head`.

$$\frac{\texttt{app}\ (\lambda x : T.e)\ v \longrightarrow e[v/x] \in \text{Dynamic Semantics} \quad \lambda \in \text{Values}}{\text{Value} \mid \text{Error} \mid \text{Dyn. Sem.} \vdash \dfrac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash \texttt{app}\ e_1\ e_2 : T_2} : \texttt{app} \mapsto \text{elim} \to}$$

$$\frac{\texttt{head}\ (\texttt{cons}\ v_1\ v_2) \longrightarrow v_1 \in \text{Dynamic Semantics} \quad \texttt{cons} \in \text{Values}}{\text{Value} \mid \text{Error} \mid \text{Dynamic Sem.} \vdash \dfrac{\Gamma \vdash e : \texttt{List}\ T}{\Gamma \vdash \texttt{head}\ e : T} : \texttt{head} \mapsto \text{elim}\ \texttt{List}}$$

Imposing that the principal argument be a value and typed at a complex type helps satisfy the progress theorem for the language being type checked. In particular, when we analyze an operator whose typing rule has been type checked with [T-ELIM] we are guaranteed to be able to apply the appropriate canonical form lemma. The meta type system for the dynamic semantics of Section 3.3 then ensures the existence of reduction rules to prove some steps, and therefore that the progress theorem would hold for the operator.

[T-ERRHANDLER] classifies the error handlers. In doing so, we make sure that *error handlers handle the error at their principal argument.* [T-ERRHANDLER] is similar to [T-ELIM]. There are two differences: it does not impose the principal argument to be typed at some precise type, and it expects a reduction rule of the form $(op_1\ \ (op_2\ \overline{args_2})\ \ \overline{args_1}) \longrightarrow exp^+$ where the operator op_2 is an error rather than a value.

To make an example, we instantiate [T-ERRHANDLER] for the typing rule (T-TRY).

$$\frac{\texttt{try}\ (\texttt{raise}\ v)\ \texttt{with}\ e \longrightarrow (e\ v) \in \text{Dynamic Semantics} \quad \texttt{raise} \in \text{Error}}{\text{Value} \mid \text{Error} \mid \text{Dyn. Sem.} \vdash \dfrac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : \top \to T}{\Gamma \vdash\ \texttt{try}\ e_1\ \texttt{with}\ e_2 : T} : \texttt{try} \mapsto \text{errorHandler}}$$

[T-DERIVED] classifies derived operators. One of their characteristics is that *derived operators do not primitively manipulate complex data, rather they pass it on to other operators.* For this reason, [T-DERIVED] checks that all the reduction rules for the operator are of the form $(op\ \overline{args}) \longrightarrow exp^+$, where $\overline{args}$ are exclusively variables. Therefore, op does not pattern-match its arguments into complex expressions. To make some examples, we instantiate [T-DERIVED] for the typing rules of `fix` and `letrec`.

$$\frac{\texttt{fix}\ v \longrightarrow v\ (\texttt{fix}\ v) \in \text{Dynamic Semantics}}{\text{Value} \mid \text{Error} \mid \text{Dynamic Sem.} \vdash \dfrac{\Gamma \vdash e : T \to T}{\Gamma \vdash \texttt{fix}\ e : T} : \texttt{fix} \mapsto \text{derived}}$$

$$\frac{\texttt{letrec}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow e_2[(\texttt{fix}\ (\lambda x.e_1))/x] \in \text{Dyn. Sem.}}{\text{Value} \mid \text{Error} \mid \text{Dyn. Sem.} \vdash \dfrac{\Gamma, x : T_1 \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash\ \texttt{letrec}\ x = e_1\ \texttt{in}\ e_2 : T_2} : \texttt{letrec} \mapsto \text{derived}}$$

[T-ERROR] handles the typing rule for the error. We enforce that the assigned type is a fresh variable T. This ensures that the error can be typed at any type. This is necessary to establish the type preservation of the language being type checked. Indeed, the error travels through contexts thanks to the rule (ERR-CTX), and it must be prepared to match the type of the expression it replaces. It is easy to see that [T-ERROR] checks (T-ERROR) successfully.

For F_{exc}, all typing rules check successfully and we end up with the classification below.

$$\text{Classification} = \{ \begin{array}{l} \lambda x : T.e \mapsto \text{value} \to, \\ \Lambda X.e \mapsto \text{value} \forall, \\ \texttt{app} \mapsto \text{elim} \to, \\ \texttt{tapp} \mapsto \text{elim} \forall, \\ \texttt{try} \mapsto \text{errorHandler}, \\ \texttt{raise}\ v \mapsto \text{error} \end{array} \}$$

We write $\text{Classification} \vdash \texttt{app} : \text{elim} \to$ for retrieving information about operators.

***Back to* Error Context.** After having meta type checked the typing rules, we can check Error Context because we have acquired knowledge on the error handler, if present. Indeed, we need to ensure **ErrCtx**, and we do so with the typing judgement Classification | Context ⊢ Error Context, defined below.

$$\boxed{\text{Classification} \mid \text{Context} \vdash \text{Error Context}}$$

$$\frac{(op\ \cdots) : \text{error} \notin \text{Classification}}{\text{Classification} \mid \text{Context} \vdash er ::= \epsilon}$$

$$\frac{\text{Classification} \vdash (op_1\ \cdots) : \text{error} \quad (op_2\ \cdots) : \text{errorHandler} \notin \text{Classification}}{\text{Classification} \mid \text{Context} \vdash \text{Context[F/E]}}$$

$$\frac{\text{Classification} \vdash (op_1\ \cdots) : \text{error} \quad \text{Classification} \vdash (op_2\ \cdots) : \text{errorHandler}}{\text{Classification} \mid \text{Context} \vdash \text{Context[F/E]} - (op_2\ F\ \cdots)}$$

If the language does not contain an error, Error Context is the empty derivation ϵ, which we consider well-typed by default. If an error is present but there are no error handlers, then we simply impose that error contexts are exactly the evaluation contexts. We recall that evaluation contexts are defined with the variable E and error contexts are defined with the variable F. With the notation Contexts[F/E] we denote the grammar of Context in which variables E are replaced with F. Ultimately, if an operator is classified as an error handler, we check that error contexts are evaluation contexts minus the error handler at the principal argument (F as the first argument of op_2).

Type checking error contexts make them synch with evaluation contexts, and helps proving the progress theorem for the language being type checked because it ensures that operators have a defined behavior if an error appears as one of their arguments.

3.3 Meta Type System for Dynamic Semantics

Figure 5 contains the meta type system for checking the dynamic semantics. The language design invariants that we enforce at this point are:

- **ValRed:** *Principal arguments, and any argument that is required to be evaluated to a value for a reduction rule to fire, must be declared as evaluation context.* For example, without the context declaration $(E\ e)$ the expression $((\lambda x.x)\ (\lambda x.x))$ `nil`) would be stuck because (BETA) would not fire and jeopardize type soundness. Similarly, since (BETA) requires that also the second argument of `app` be a value, missing the evaluation context would jeopardize type soundness.
- **AllVal:** *Each elimination form of some type manipulates all the values of that type.* For example, were we to miss to specify the reduction rule for the predecessor operation (`pred zero`) $\longrightarrow$ `error`, the expression (`pred zero`) would be stuck, jeopardizing type soundness.
- **ErrSucc:** *Error handlers must handle both the error and the case of success.* For example, if we were to define (R-TRY-ERROR) as the sole behavior of the error handler, the expression `try true with` e would be stuck, jeopardizing type soundness. A reduction rule such as (R-TRY-SUCCESS), which expects a value at principal argument position, must be specified.
- **Preserv:** Naturally, reductions must be type preserving, i.e., in a reduction $e \longrightarrow e'$, e and e' must have the same type.

[R-MAIN] is the main typing rule. First, it imposes that (CTX) and (ERR-CTX) are present. Next, it makes sure that the rest of the reduction rules $\phi_1, \ldots, \phi_n$ are well-typed.

Meta type checking a reduction rule returns a *move map* (C), which is a description of the computational step that the reduction rule enables. A move map assigns a *move* (M) to the operator that is the subject of the reduction rule. Moves can be:

- eliminates op: for reduction rules of elimination forms, which manipulate some specific value.
- handlesError: for the reduction rule that handles the error.
- handlesSuccess: for the error handler to handle the case when an error does not occur.
- moves: for reduction rules of derived operators.

When all the reduction rules have been checked, [R-MAIN] collects all the move maps and we perform an exhaustiveness check to ensure that **AllVal** and **ErrSucc** are fulfilled. This is done with Moves `exhausts` Classification.

[R-ELIMINATION] handles the case of reduction rules for elimination forms. The first aspect that [R-ELIMINATION] enforces is the language design pattern ***Elim***. To this aim, we check that op_1 is an elimination form for a type c and that op_2 is a value of the same type c. Next, [R-ELIMINATION] covers ***ValRed***. In particular, we check that the principal argument (position 1) is an evaluation context. Not only that, we check that any other argument of op_1 that is required to be a value has an evaluation context defined. These checks help satisfy the progress theorem of the language being type checked. If some arguments need to become a value for the rule to fire, the existence of the corresponding evaluation contexts let them evaluate indeed. Also, [R-ELIMINATION] ensures that when the principal argument becomes the value op_2 then we can take a step with this reduction rule, satisfying the progress theorem.

Lastly, [R-ELIMINATION] and all other rules of Figure 5 check that the overall reduction is type preserving. This is done with the last line of the premises. We postpone the discussion of type preservation until Section 3.4.

To make an example, we instantiate [R-ELIMINATION] for (BETA) and one of the reduction rules for `head`. (Below and throughout this subsection we have elided the type preservation check, which we discuss later).

$$\frac{\begin{array}{l}\text{Classification} \vdash \texttt{app} : \text{elim} \longrightarrow \\ \text{Classification} \vdash \lambda\ x : T.e : \text{value} \longrightarrow \\ \text{Context} \vdash \texttt{app} : \{1, 2\}\end{array}}{\begin{array}{lll}\text{Context} & & \texttt{app} \\ \mid \text{Classif.} & \vdash (\texttt{app}\ (\lambda\ x : T.e))\ v \longrightarrow e[v/x] : & \mapsto \\ \mid \text{Type System} & & \text{eliminates } \lambda\end{array}}$$

$$\frac{\begin{array}{l}\text{Classification} \vdash \texttt{head} : \text{elim List} \\ \text{Classification} \vdash \texttt{cons}\ v_1\ v_2 : \text{value List} \\ \text{Context} \vdash \texttt{head} : \{1\}\end{array}}{\begin{array}{lll}\text{Context} & & \texttt{head} \\ \mid \text{Classif.} & \vdash \texttt{head}\ (\texttt{cons}\ v_1\ v_2) \longrightarrow v_1 : & \mapsto \\ \mid \text{Type System} & & \text{eliminates cons}\end{array}}$$

Notice that a call-by-name application with reduction rule $(\texttt{app}\ (\lambda\ x : T.e_1))\ e_2 \longrightarrow e_1[e_2/x]$ would be handled similarly, with the difference that the check on evaluation contexts would be $\text{Context} \vdash \texttt{app} : \{1\}$. This is because, roughly speaking, the second argument of the application is an e rather than a v.

[R-ERRHANDLER] addresses the reduction rule for error handling. This rule is similar to [R-ELIMINATION]. We first detect that the operator at hand is an error handler, and then we check that the principal argument forms an error. To make an example, [R-ERRHANDLER] is instantiated for

Move Maps C ::= $op \mapsto M$
Moves M ::= eliminates op | handlesError | handlesSuccess | moves

Moves `exhausts` Classification whenever

$\{op_1 : \text{elim } c,\ op_2 : \text{intro } c\} \subseteq \text{Classification}$ implies $op_1 :$ eliminates $op_2 \in$ Moves, and
$\{op_1 : \text{errorHandler},\ op_2 : \text{error}\} \subseteq \text{Classification}$ implies $\{op_1 : \text{handlesError}, op_1 : \text{handlesSuccess}\} \subseteq$ Moves.

Context | Classification | Type System ⊢ Dynamic Semantics

[R-MAIN]

$$\frac{\begin{array}{c} ErrCtx = \begin{cases} \{(\text{ERR-CTX})\}, & \text{if Classification} \vdash op : \text{error} \\ \emptyset, & \text{otherwise} \end{cases} \\ \text{Context} \mid \text{Classification} \mid \text{Type System} \vdash \phi_1 : M_1 \\ \vdots \\ \text{Context} \mid \text{Classification} \mid \text{Type System} \vdash \phi_n : M_n \\ M_1, \ldots, M_n \ \texttt{exhausts} \ \text{Classification} \end{array}}{\text{Context} \mid \text{Classification} \mid \text{Type System} \vdash ErrCtx \cup \{(\text{CTX}), \phi_1, \ldots, \phi_n\}}$$

[R-ELIMINATION]

$$\frac{\begin{array}{ll} \text{Classification} \vdash op_1 : \text{elim } c & \text{Context} \vdash op_1 : contextPositions \\ \text{Classification} \vdash (op_2\ \overline{args'}) : \text{value } c & \forall i \in \{1\} \cup \{i \mid args_i = v\}, i \in contextPositions \\ \text{Type System} \vdash_{\text{lhs}} (op_1\ (op_2\ \overline{args'})\ \overline{args}) : ty \Rightarrow \Gamma^s & \text{Type System} \mid \Gamma^s \vdash_{\text{rhs}} exp^+ : ty \end{array}}{\text{Context} \mid \text{Classification} \mid \text{Type System} \vdash (op_1\ (op_2\ \overline{args'})\ \overline{args}) \longrightarrow exp^+ : op_1 \mapsto \text{eliminates } op_2}$$

[R-ERRHANDLER]

$$\frac{\begin{array}{ll} \text{Classification} \vdash op_1 : \text{errorHandler} & \text{Context} \vdash op_1 : contextPositions \\ \text{Classification} \vdash (op_2\ \overline{args'}) : \text{error} & \forall i \in \{1\} \cup \{i \mid args_i = v\}, i \in contextPositions \\ \text{Type System} \vdash_{\text{lhs}} (op_1\ (op_2\ \overline{args'})\ \overline{args}) : ty \Rightarrow \Gamma^s & \text{Type System} \mid \Gamma^s \vdash_{\text{rhs}} exp^+ : ty \end{array}}{\text{Context} \mid \text{Classification} \mid \text{Type System} \vdash (op_1\ (op_2\ \overline{args'})\ \overline{args}) \longrightarrow exp^+ : op_1 \mapsto \text{handlesError}}$$

[R-SUCCESS]

$$\frac{\begin{array}{ll} \text{Classification} \vdash op : \text{errorHandler} & \text{Context} \vdash op : contextPositions \\ & \forall i \in \{1\} \cup \{i \mid args_i = v\}, i \in contextPositions \\ \text{Type System} \vdash_{\text{lhs}} (op\ v\ \overline{args}) : ty \Rightarrow \Gamma^s & \text{Type System} \mid \Gamma^s \vdash_{\text{rhs}} exp^+ : ty \end{array}}{\text{Context} \mid \text{Classification} \mid \text{Type System} \vdash (op\ v\ \overline{args}) \longrightarrow exp^+ : op \mapsto \text{handlesSuccess}}$$

[R-MOVES]

$$\frac{\begin{array}{ll} \text{Classification} \vdash op : \text{derived} & \text{Context} \vdash op : contextPositions \\ & \forall i \in \{1\} \cup \{i \mid args_i = v\}, i \in contextPositions \\ \text{Type System} \vdash_{\text{lhs}} (op\ \overline{args}) : ty \Rightarrow \Gamma^s & \text{Type System} \mid \Gamma^s \vdash_{\text{rhs}} exp^+ : ty \end{array}}{\text{Context} \mid \text{Classification} \mid \text{Type System} \vdash (op\ \overline{args}) \longrightarrow exp^+ : op \mapsto \text{moves}}$$

Figure 5. Meta Type system: Dynamic Semantics

(R-TRY-ERROR) in the following way.

$$\frac{\begin{array}{l} \text{Classification} \vdash \texttt{try} : \text{errorHandler} \\ \text{Classification} \vdash (\texttt{raise}\ v) : \text{error} \\ \text{Context} \vdash \texttt{try} : \{1\} \end{array}}{\begin{array}{l} \text{Context} \\ \mid \text{Classification} \\ \mid \text{Type System} \end{array} \vdash \begin{array}{l} \texttt{try raise } v \\ \texttt{with } e \end{array} \longrightarrow (e\ v) \quad \begin{array}{c} \texttt{try} \\ \mapsto \\ \text{handlesError} \end{array}}$$

[R-SUCCESS] addresses the reduction rule that specifies the behavior of the error handler when an error does not occur. We first detect that the operator is an error handler. Then, we check that a value v appears in principal argument position. To make an example, [R-SUCCESS] checks (R-TRY-SUCCESS) in

the following way.

$$\frac{\begin{array}{c}\text{Classification} \vdash \texttt{try} : \text{errorHandler} \\ \text{Context} \vdash \texttt{try} : \{1\}\end{array}}{\begin{array}{ll}\text{Context} & \texttt{try} \\ \mid \text{Classification} \vdash \texttt{try}\ v\ \texttt{with}\ e \longrightarrow v : & \mapsto \\ \mid \text{Type System} & \text{handlesSuccess}\end{array}}$$

[R-MOVES] handles the reduction rule of derived operators. To make an example, the reduction rules for `fix` and `let` are handled as follows. (Evaluation contexts are `fix` $E \mid$ `let` $x = E$ `in` e.)

$$\frac{\text{Classification} \vdash \texttt{fix} : \text{derived} \qquad \text{Context} \vdash \texttt{fix} : \{1\}}{\begin{array}{l}\text{Context} \\ \mid \text{Classification} \vdash \texttt{fix}\ v \longrightarrow v\ (\texttt{fix}\ v) : \texttt{fix} \mapsto \text{moves} \\ \mid \text{Type System}\end{array}}$$

$$\frac{\text{Classification} \vdash \texttt{let} : \text{derived} \qquad \text{Context} \vdash \texttt{let} : \{1\}}{\begin{array}{l}\text{Context} \\ \mid \text{Classification} \vdash \texttt{let}\ x = v\ \texttt{in}\ e \longrightarrow e[v/x] : \texttt{let} \mapsto \text{moves} \\ \mid \text{Type System}\end{array}}$$

3.4 Meta Type Checking for Type Preservation

A mandatory part of type soundness is that reduction rules must be type preserving. Given a reduction rule $exp \longrightarrow exp'$, we have to ensure that the types of exp and exp' coincide. This is done with the last line of premises in each rule of Figure 5. Differently from the rest of the meta type system, where organizational principles play a role, we are unaware of language organization methods that ensure type preservation. Language designers typically simply write reduction rules making sure that the type on the left-hand side and that on the right-hand side coincide. Works such as [Pfenning and Schürmann 1999; Schürmann 2000] and [Grewe et al. 2015] have demonstrated that type preservation checks can be automated with theorem provers. Our approach adapts that method and integrates it within the formalism of our meta type system.

Ideally, we need to check that for all Γ, $\Gamma \vdash exp : T$ implies $\Gamma \vdash exp' : T$, but checking all possible type environments is prohibitive. Therefore, we approximate such a check with the use of a *symbolic type environment*, which contains the assumptions on the types of the variables. Our first check is Type System $\vdash_{\text{lhs}} exp : ty \Rightarrow \Gamma^s$, which means that according to the type system we have that exp has type ty under assumptions Γ^s, which are computed as output. The way we build Γ^s for exp is by inverting the typing rules used to type check exp. As we essentially use the method in [Grewe et al. 2015; Pfenning and Schürmann 1999; Schürmann 2000] we omit this part here, though it can be found in Appendix B.

To make an example, `head (cons` v_1 v_2`)` $\longrightarrow v_1$ gives $\Gamma^s = \Gamma \vdash v_1 : T, \Gamma \vdash v_2 : \texttt{List}\ T$. We also obtain its type T. Then, we check Type System $\mid \Gamma^s \vdash_{\text{rhs}} exp' : ty$, which means that exp' has type ty according to the type system and Γ^s, which is an input this time. This check solves a provability problem:

$$\begin{array}{l}\text{Type System} \cup \text{Lemmas} \cup \\ \Gamma \vdash v_1 : T, \Gamma \vdash v_2 : \texttt{List}\ T\end{array} \models \Gamma \vdash v_1 : T.$$

That is, the formula $\Gamma \vdash v_1 : T$ can be proved with the inference system formed by the type system, substitution lemmas (Lemmas, see below), and the formulae in Γ^s.

Lemmas contains the well-known substitution lemmas:

$$\text{(TYP-SUB1)}\ \frac{\Gamma, x : T_1 \vdash e : T_2 \qquad \Gamma \vdash e' : T_1}{\Gamma \vdash e[e'/x] : T_2} \qquad \text{(TYP-SUB2)}\ \frac{\Gamma, X \vdash e : T}{\Gamma \vdash e[T'/X] : T[T'/X]}$$

We have proved that these lemmas automatically hold for any given language that is restricted to our setting.

4 Correctness of our Meta Type System

We have proved the correctness of our meta type system: Language definitions that are well-typed are guaranteed to be type sound.

Theorem 4.1 (Correctness of $\vdash \mathcal{L}$)**.**
Given a language definition $\mathcal{L}$, if $\vdash \mathcal{L}$ then $\mathcal{L}$ is type sound.

The proof is in Appendix A. Once the language definition is type checked, it has a predictable structure, and we can automate the proof of Wright and Felleisen [Wright and Felleisen 1994]. The proof replays canonical form lemmas, the progress theorem, type preservation theorem and, ultimately, the type soundness theorem.

5 Evaluation

We have implemented our meta type system in a tool called LANG-N-CHECK [Cimini 2015]. This tool is written in OCaml and type checks language definitions that are written in a rather intuitive domain-specific language. An example of language definition is in Figure 1 of the introduction section.

The type preservation checks for reduction rules are done by compiling languages into the Abella theorem prover [Baelde et al. 2014] and by running the queries described in Section 3.4.

Deriving Type Soundness of Languages. We have used LANG-N-CHECK to derive the type soundness of several languages: STLC, STLC with integers, booleans, pairs, lists, sums, tuples, `fix`, `let`, `letrec`, unit, universal types, recursive types, option types, exceptions, list operations such as append, map, mapi (also depends on the position in the list of the current element being processed), filter, filteri, range, list length, reverse, and the recursor of natural numbers (natrec). We have also considered different evaluation strategies for the features mentioned above: call-by-value, call-by-name and a parallel reduction strategy (both function and argument can evaluate non-deterministically), as well as lazy pairs, lazy lists, and lazy tuples, and both left-to-right and right-to-left evaluations.

As we said above, LANG-N-CHECK compiles languages into the Abella theorem prover. If they type check successfully LANG-N-CHECK also generates their machine-checked proof of type soundness. We therefore have a mechanized proof for all the languages that we have tested. This gives us high confidence that, besides the theoretical guarantees of our paper, also our implementation is reliable.

Some remarks on the type preservation checks and the generation of the machine-checked proof are in order. These are two different processes. During type checking, we use Abella to run queries that check for type preservation. After type checking, LANG-N-CHECK generates the proof code that describes the proof of type preservation. This code contains instructions to do case analysis on hypothesis at the correct order, apply the inductive hypothesis, and so on. If the mentioned queries succeed during type checking, then the proof code succeeds, too, as the proof is guaranteed to find what it needs during its execution.

Informative Error Messages. LANG-N-CHECK has been used to teach programming languages theory [Cimini 2019]. That paper remarks that LANG-N-CHECK can report informative error messages to the user.

We provided an example in the introduction section. To show other examples[3], consider the listing in Figure 1 of the introduction.

- Were we to miss one of the reduction rules for the `if` operator, say the rule at line 22, LANG-N-CHECK would reject the language definition and print the error message *"Operator **if** is elimination form for the type **bool** but does not have a reduction rule for handling one of the values of type **bool**: value **tt**"*.
- Were we to miss the context declaration (`app` v C) at line 6, LANG-N-CHECK would reject the language definition and print the error message *"Reduction rule for elimination form **app** requires argument 2 to be a value but that argument is not declared as evaluation context, hence some programs may get stuck"*.
- If the language definition had contexts declarations (`app` C v) and (`app` v C), LANG-N-CHECK would reject the definition and print the error message *"Evaluation contexts have cyclic dependencies, hence some programs may get stuck"*.

Limitations and Future Work. There are type sound languages whose type soundness cannot be verified with our system. We shall discuss some limitations of our meta type system.

The shape of the typing relation and reduction rules are enforced to be $\Gamma \vdash e : T$ and $e \longrightarrow e$, respectively, which is standard for purely functional languages, and does not capture languages with effects, typestate, and stores, to name a few. We plan to address those features in the future. We do not have a general treatment for subtyping and type-based access, therefore calculi such as Featherweight Java [Igarashi et al. 2001] are currently out of the scope of LANG-N-CHECK. Similarly, the grammar for types forbids dependent types and refinement types. These are advanced features that we leave for future work. We also plan to extend our work to give the user the possibility to define layered grammars such as expressions vs statements.

Languages are restricted to an organization that is based on introduction forms and elimination forms, as well as other roles. Some calculi may not fit this schema even within the realm of functional languages. For example, the gradually typed λ-calculus has a cast operator that is some times a value and some times performs computational steps [Siek and Taha 2006], and therefore defies this schema.

Our setting only allows unary binding [Cheney 2005]. This has been sufficient for the languages that we have checked. However, such restricted approach leaves out some languages, especially those that are non-lexically scoped. We plan to adopt more sophisticated binding structures. In this regard, we are looking at integrating scopes and frames [Néron et al. 2015; Poulsen et al. 2016], which have been successfully employed in other systems [Poulsen et al. 2018].

6 Related Work

Intrinsically Typed Languages. Poulsen et al have demonstrated that intrinsic typing can derive the soundness of a large subset of Middleweight Java [Poulsen et al. 2018]. The applicability of their work is much more general than that of this paper in at least two aspects. First, they capture a rather complex language with a store, which is out of the scope of our meta type system. Second, they employ a more general binding structure to accommodate classes. We make use of a less expressive unary binding approach.

The title of our paper is inspired by theirs[4]. 1) We use extrinsic typing rather than intrinsic typing. As we have explained in the introduction, the two are conceptually different approaches. Also, our meta type system can print error messages that use the jargon of language designers, which is not the case for [Poulsen et al. 2018].

2) We work with operational semantics formulations rather than definitional interpreters. The latter are Agda implementations of languages, whose evaluator must be augmented with the so called *fuel* to help the compiler derive termination [Owens et al. 2016]. In contrast, our meta type system may appeal a part of the community for working with language definitions that are in 1-1 correspondence with pen&paper formulations. The syntax of LANG-N-CHECK is indeed inspired by that of the much earlier work of Ott [Sewell et al. 2007]. Cimini [Cimini 2019] observes that students

[3] Similar examples are made in Cimini 2019.

[4] For ease of reference, the title is "Intrinsically Typed Definitional Interpreters for Imperative Languages".

could use LANG-N-CHECK after having been taught type systems and operational semantics with the TAPL textbook [Pierce 2002]. Moreover, Poulsen et al. target a big-step semantics, which does not need evaluation contexts. In contrast, we use small-step semantics and our meta type system contributes a formal treatment for evaluation contexts.

3) Our title makes it clear that the scope of our paper is that of functional languages rather than imperative languages. In this regard, we acknowledge that the state of the art of intrinsic typing is substantially ahead. This is further exemplified by the existence of work on linear and session types with intrinsic typing [Rouvoet et al. 2020], which also are out of our scope.

Automated Theorem Proving. The seminal work of Schürmann and Pfenning shows that the type soundness of non-trivial functional languages can be automatically established in the LF-based theorem prover Twelf [Pfenning and Schürmann 1999; Schürmann 2000]. The Veritas tool automates the verification of the type soundness of languages by compiling them into a first order theorem prover, and by checking that suitable formulae hold [Grewe 2019; Grewe et al. 2016, 2017, 2015]. The way our meta type system checks type preservation stems directly from these works, but our approach to progress is different. Differently from these works, we establish progress by encoding language invariants as a typing discipline rather than querying a theorem prover with formulae. To our knowledge, those tools also do not report error messages that refer to language design terminology, such as the messages in Section 5.

Type Sound Language Extensions. Schwaab and Siek [Schwaab and Siek 2013], and Delaware et al [Delaware et al. 2013] provide solutions to the composition of already existing proofs of type soundness. Lorenzen and Erdweg's work on SoundX [Lorenzen and Erdweg 2016] proposes a method to establish the soundness of language extensions, making sure that their desugaring is correct. In contrast to these works, our meta type system checks that a language definition given from scratch is type sound in the first place. These works are somehow orthogonal to ours, and we plan to integrate their insights in our context.

Marino and Millstein propose a generic type-and-effect system [Marino and Millstein 2009] that language designers can instantiate with effects. The resulting languages are guaranteed to be type sound. However, the underlying language of the framework is fixed and only the effects of the language are parametrized. Therefore, adding new operators does not come with a guarantee of type soundness. In contrast, our work allows language designers to define the operations of languages and check for type soundness. In the future, we plan on integrating the approach of Marino and Millstein into LANG-N-CHECK and add a now missing treatment for effects.

Model Checking. Roberson et al. [Roberson et al. 2008] propose a model checking approach to type soundness in which configurations are generated, steps are computed, and then type checked. Similarly to testing, this approach can detect bugs but cannot guarantee type soundness.

Language Workbenches and Semantics Engineering Tools. There are several tools that support the specification of languages, such as Ott [Sewell et al. 2007], Lem [Mulligan et al. 2014], the K framework [Roşu and Şerbănuţă 2010], and PLT Redex [Felleisen et al. 2009]. Furthermore, language workbenches also assist language designers with a plurality of language services [Erdweg et al. 2013, 2015; Fowler 2005], and some have extensive support for implementing type systems, such as MPS [Voelter and Solomatov 2010], Xtext [Bettini 2011; Efftinge and Spönemann [n.d.]], and SugarJ [Erdweg et al. 2011], to name a few. However, these systems do not establish the type soundness of the languages being defined[5].

A related work in the context of tools for teaching is SASyLF [Aldrich et al. 2008], a proof assistant that has been used in the classroom. Students can write derivations and proofs in readable proof code and SASyLF provides feedback on students' mistakes. Differently from SASyLF, students do not write proofs in LANG-N-CHECK, they receive feedback directly on the language definitions they provide.

7 Conclusions

In this paper, we have presented an extrinsic type system that establishes the type soundness of functional languages defined in operational semantics. Our meta type system is based on a typing discipline that models a high-level language design organization. We have proved that our type system is correct. Furthermore, we have implemented our work in the LANG-N-CHECK tool and we have applied it to a plethora of functional languages.

Intrinsic typing has certainly demonstrated that it can scale well, thanks to recent advances in its theory and practice [Poulsen et al. 2018; Rouvoet et al. 2020]. We have informally referred to such advances as **Stage 2** in the introduction. This paper introduces the extrinsic typing approach to type soundness and we believe that it helps to establish **Stage 1** for this research line. In the future, we plan to tackle sophisticated languages with extrinsic typing.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1763922.

[5] A plan to equip Spoofax [Kats and Visser 2010] with verification capabilities has been reported [Visser et al. 2014] but we are not aware of any integrated component for it yet.

A Correctness of the Meta Type System

We follow the syntactic approach to type soundness of Wright and Felleisen [Wright and Felleisen 1994]. We generate canonical form lemmas, progress, type preservation and, ultimately, type soundness theorems and proofs.

A.1 Progress Theorem

PROGRESS THEOREM:
If $\vdash \mathcal{L}$ then
if $\emptyset \vdash e : T$ then e progresses.

An expression e *progresses* whenever either e is a value, e is an error, or there exists e' such that $e \longrightarrow e'$.

The Main Progress Theorem. Assume $\vdash \mathcal{L}$ and $\vdash e : T$. The proof is by induction on $\vdash e : T$. As $\vdash e : T$ is provable, it means that there exists a typing rule ϕ of the form $\dfrac{f_1, \dots f_n}{conclusion}$ that is 'satisfied', that is, there exists a substitution γ from logical variables (of the rule) to terms such that $f_i\gamma$ are all satisfied for all f_i, and $(conclusion)\gamma = \vdash e : T$.

For each case of the induction on $\vdash e : T$, the proof goes as follows. Since $\vdash \mathcal{L}$ and ϕ is a typing rule of $\mathcal{L}$ then ϕ has been type checked. Therefore, e has a top level operator op and arguments, as this is the shape imposed for typing rules. It also means that *all* the arguments of the operator, say $\widetilde{e}$, are the subject of a typing premise, because our type checker imposes that shape for typing rules. Therefore we can apply the inductive hypothesis on the arguments $\widetilde{e}$. In particular, we apply the inductive hypothesis for those arguments e_i with index $i \in contextPosition$ from Context $\vdash op : contextPosition$, i.e., only to the contextual arguments. As Context has been meta type checked, e_i is not under a binder. Therefore the typing premise for e_i is of the form $\Gamma \vdash e_i : ty_i$ (and not with $\Gamma, x : T$). Therefore we can apply the inductive hypothesis to $\vdash e_i\gamma : ty_i\gamma$ for all of these, and we derive that $e_i\gamma$ *progresses*. We call the Progress Lemma for op (defined below), which expects exactly those progress assumptions $e_i\gamma$ *progresses*. The Progress Lemma for op satisfies the Main Progress Theorem, as discussed below.

Progress Lemma for op**.** Fix an operator op in $\mathcal{L}$, and let $Context \vdash op : \{1, \dots, n\} \in$ Context. Without loss of generality, here contextual arguments appear first as a presentational aid. The progress theorem for op is: (We mark one of the assumption with (H)).

if $\vdash \mathcal{L}$ and (H) $\vdash (op\ e_1 \dots e_n\ \widetilde{e}) : T$, and *progress* $e_1, \dots$, *progress* e_n then *progress* $(op\ e_1 \dots e_n\ \widetilde{e})$.

The proof is by case analysis on (H). If $\vdash (op\ e_1 \dots e_n\ \widetilde{e}) : T$ has been proved, than it has been proved with a typing rule ϕ. We do case analysis on all *progress* $e_1, \dots$ *progress* e_n, but in a suitable order. Since $\vdash \mathcal{L}$, we have that `acyclic`(Context). Therefore we can choose an order for argument positions for which the following invariant (**invariant**) holds: we do case analysis on progress e_i before the case analysis on progress e_j if the context for i-th argument of op does not depend on the valuehood of the argument j of op. As this order must exist, without loss of generality, to aid the proof we fix that the evaluation contexts are left-to-right. After the series of cases analysis on progress e_i, we are at the leftmost child of the leftmost tree of the cases.

Example: If we have two arguments, the first case analysis on *progress* e_1 opens three cases: 1) *value* $e_1 \wedge$ *progress* e_2, 2) *step* $e_1 \wedge$ *progress* e_2, 3) *error* $e_1 \wedge$ *progress* e_2, where *step* e means that e can take a step, and *error* e means that e is an error. After we open this case analysis we are at the left child 1). We now do case analysis on *progress* e_2 and we open other three cases only on the left child: the leftmost subtree is 1) *value* $e_1 \wedge$ *value* e_2, 2) *value* $e_1 \wedge$ *step* e_2, 3) *value* $e_1 \wedge$ *error* e_2. Afterwards, we are still at the leftmost child: *value* e_1 and *value* e_2, thanks to the invariant.

We show only the case for this leftmost child here, as the further cases analysis generated follow a similar reasoning.

The proof is by case analysis on the role that op has in Classification. This role exists because $\vdash \mathcal{L}$, and therefore a typing rule for op has been type checked, then Classification has been produced.

- op is (value c): As Classification $\vdash op\ args$: value c then $op\ args \in$ Values. There are three cases. The first is when we strive to apply the value definition for op, however, this definitions may use a v variable if some argument is required to be a value for the definition to apply. As we have type checked op, these arguments must be in *contextPosition* and therefore they all are involved in a *progress* $e_1, \dots$, *progress* e_n assumptions. We are also in the case in which we have done a case analysis on all of them, and we are in the case where all of them are values, and so the definition applies.
  We have
  - **STEP:** case $e_j \longrightarrow e'_j$. Moreover, we have that all arguments of op before j are all values. Therefore we can apply the evaluation context $(op\ \widetilde{v}\ E_j\ \widetilde{e})$ (recall that we assume left-to-right evaluation but the argument generalizes for any topological sort on the dependencies of contexts) to prove a step $(op\ \widetilde{v}\ e_j\ \widetilde{e}) \longrightarrow (op\widetilde{v}\ e'_j\ \widetilde{e})$, and so we progress.
  - **ERR:** e_j is an error. Moreover, we have that all arguments of op before j are all values. Therefore we can apply the evaluation context $(op\ \widetilde{v}\ E_j\ \widetilde{e})$. Since we have type checked Error Context(op) and we are in the context of a value $(op\ args$: value $c)$ (rather than an error handler), we have that $(op\ \widetilde{v}\ F_j\widetilde{e})$ is an error context. Therefore, we can prove a step $(op\ \widetilde{v}\ e_j\ \widetilde{e}) \longrightarrow e_j$, i.e. a step to the error, with CTX-ERR. So, we progress.

- op is (elim c): Then [T-ELIM] has meta type cheked ϕ. This means that the ϕ has a typing premise for the principal argument of the form $\vdash\ e_1 : (c\ \widetilde{T})$. Also, we are exploring the case where e_1 is a value. Therefore, we apply the Canonical Forms Lemma for c (described in the following paragraph). This means that $e_1 = t_1 \vee \ldots \vee t_m$ (this is notation from the next paragraph) with all t_k be with form $(op_2\ args)$ and $op_2 \in$ Values. Then, since Dynamic Semantics has been meta type checked, it must have produced Moves that passed the exhaustiveness check `exhaust`. Therefore we have op : elim c and Classification(op_2) = value c, which enforces that a reduction rule of the form $r = (op\ (op_2\ \overline{ev_2})\overline{ev_1}) \longrightarrow exp$ exists. As we are in the case where the principal argument is a value, and also all other argument in $\overline{ev_1}$ are values, we have that this reduction rule fires and takes a step, and so we progress.
  The other cases for op : elim c are proved as in **STEP** and **ERR** above.

- op is (error): This is handled similarly as to values, though it satisfies progress because the error definition applies rather than a value definitions.
  The other cases for op : elim c are proved as in **STEP** and **ERR** above.

- op is (derived): Then ϕ has been meta type checked by [T-DERIVED]. Therefore, a rule $(op\ \widetilde{v_i}\ \widetilde{e_i'}) \longrightarrow e_i$ exists. Also, all those v_i are in evaluation contexts, and so have been involved in the progress hypothesis above and a case analysis on all of them is being performed. So we are in the case where they are all values and thus the reduction rule can apply because our type checker imposes that the shape of the reduction rule does *not* pattern-match arguments for the derived operator op, but imposes that it requires the arguments to be vs, and we are in the case where those are all values indeed. So the rule fires, and we progress.
  The other cases for op : elim c are proved as in **STEP** and **ERR** above.

- op is (errHandler): Then ϕ has been meta type checked by [T-ERRHANDLER]. Then, since Dynamic Semantics has been meta type checked, it must have produced Moves, which by our exhaustiveness check it contains op : handlesSuccess. This means that it exists a reduction rule of the form $(op\ v\ \overline{ev}) \longrightarrow exp$. We are in the case where the arguments are values, and so they satisfy that v in first argument position. So this rule fires, and we progress. Next. we need to prove the cases for *step* and *error*. The former is handled with **STEP**. The latter is handled differently from the previous cases as type checking of Error Context excluded an error context for that position. However, exhaustiveness of Moves imposes that we have op : handlesError, which in turn, implies that there exists a reduction rule that fires when the error is at that position. Therefore, this reduction rule can apply and prove a step, and so we progress.

Canonical Forms Lemma for c**.** In the theorem below, we have marked an assumption with (H) and another with (V).

Theorem A.1. *if* $\vdash \mathcal{L}$ *and (H)* $\vdash\ e : (c\ \widetilde{T})$ *and (V) value* e *then* $e = OR(\{(op\ args) \mid Classification \vdash (op\ args) : value\ c\})$.

We use OR to denote the disjunction of a set of formulae. *value* e means that e can be derived with the grammar for Value in $\mathcal{L}$.

The proof is by case analysis on (H). If $\vdash\ e : (c\ \widetilde{T})$ has been proved, than it has been proved with a typing rule ϕ.

As Type System has been meta type checked, the conclusion of the typing rule has a form of an expression with a top level operator op.

We reason by cases analysis on the classification of op. if Classification $\vdash$ $(op\ args)$: value c) (notice the same c), then it is one of the values in $OR(\{(op\ args) \mid$ Classification $\vdash$ $(op\ args)$: value $c\})$, which proves this case. The case Classification $\vdash$ $(op\ args)$: value $c'\})$ with $c' \neq c$ cannot happen because T-VALUE imposes that $(op\ args)$ is typed at a type $(c'\ \widetilde{T})$ but we are doing a case analysis on assumption (H) which is $\vdash\ e : (c\ \widetilde{T})$, hence the case analysis cannot give a case with $c' \neq c$.

If the operator op is such that Classification $\vdash$ $(op\ args)$: elim c, or derived c, or error c, or errorHandler c, then there is a contradiction with (V), so these cases are discarded, and are proved successfully.

A.2 Type Preservation

TYPE PRESERVATION THEOREM :
if $\vdash \mathcal{L}$ then
for all expressions e, e' and types T,
if $\emptyset \vdash\ e : T$ and $e \longrightarrow e'$ then $\emptyset \vdash\ e' : T$

The proof is by induction on $e \longrightarrow e'$.

Since $e \longrightarrow e'$ is provable, then there exist a reduction rule ϕ with conclusion $concl = exp \longrightarrow exp'$ and a substitution γ such that $concl\gamma = e \longrightarrow e'$.

We have that ϕ has been type checked, therefore our type checker has classified a move for ϕ such as eliminates op, or some other move. We show the case when the role of ϕ is eliminates op because the cases for the other moves are subsumed by this. In particular, error handlers follow the same line, and derived operators are even simpler as they do not have a nested expression that is pattern-matched.

We have to prove that $(exp')\gamma$ is of type T.

As we have type checked ϕ as eliminates op, we know that $e = (op\ (op_2\ \widetilde{args1})\ \widetilde{args2})$, as this is the shape required by [R-ELIM]. We have $\emptyset \vdash e : T$ from the proviso of the theorem, which must have been proved with a typing rule of the form $\frac{ps}{\Gamma \vdash (op\ ...)}$. Here, ps are the premises that contribute to populate Γ^s for the type preservation check. As the typing rule has been type checked, too, its shape must be such that all the arguments that are expressions are recursively type checked. We therefore have a premise in ps that type checks $(op_2\ \widetilde{args1})$, and this formula has been proved, too. Therefore there is another typing rule that has type checked that formula, and has the form $\frac{ps'}{\Gamma \vdash (op_2\ ...)}$, where ps', too, contributes to populate Γ^s. Now, since [R-ELIM] has been applied, also its type preservation checks have been satisfied, and we have that (#) Type System $\cup$ Lemmas $\cup\ \Gamma^s \models exp : T$ and that (*) Type System$\cup$Lemmas$\cup\Gamma^s \models exp' : T$. The variables of exp and exp' have been assigned a type in Γ^s. The rule meta-variables of exp and exp' are also in the domain of γ (the substitution that has proved [R-ELIM]). As (#) and (*) have been proved with those variables universally quantified, any expression that we substitute to those still makes (#) and (*) provable. Therefore, we have (#) Type System$\cup$Lemmas$\cup$ $\Gamma^s \models (exp)\gamma : T$ and (*) Type System $\cup$ Lemmas $\cup\ \Gamma^s \models (exp')\gamma : T$. As $e = (exp)\gamma$ and $e' = (exp')\gamma$, then the LHS and RHS have the same type.

Contextual rules and error context rules follow standard reasoning (recall that errors are always typed at any type).

A.3 Type Soundness

Type soundness follows from progress and preservation in the usual way.

B Inverting Typing Rules for Type Preservation

(INVERT-ONE)

$$\frac{\begin{array}{c}\text{Type System}._{\text{premises}}((op\ \widetilde{args})) = \Gamma^s \\ \begin{array}{l}\text{Type System} \\ \cup\ \text{Lemmas} \\ \cup\ \Gamma^s\end{array} \models (op_1\ (op_2\ \widetilde{args1})\ \widetilde{args2}) : ty\end{array}}{\text{Type System} \vdash_{\text{lhs}} (op\ \widetilde{args}) : ty \Rightarrow \Gamma^s}$$

(INVERT-TWO)

$$\frac{\begin{array}{c}\text{Type System}._{\text{premises}}((op_2\ \widetilde{args1})) = \Gamma_1^s \\ \text{Type System}._{\text{premises}}((op_1\ (op_2\ \widetilde{args1})\ \widetilde{args2})) = \Gamma_2^s \\ \Gamma^s = \Gamma_1^s \cup \Gamma_2^s \\ \begin{array}{l}\text{Type System} \\ \cup\ \text{Lemmas} \\ \cup\ \Gamma^s\end{array} \models (op_1\ (op_2\ \widetilde{args1})\ \widetilde{args2}) : ty\end{array}}{\text{Type System} \vdash_{\text{lhs}} (op_1\ (op_2\ \widetilde{args1})\ \widetilde{args2}) : ty \Rightarrow \Gamma^s}$$

(RHS-CHECK)

$$\frac{\text{Type System} \cup \text{Lemmas} \cup \Gamma^s \models exp : ty}{\text{Type System} \mid \Gamma^s \vdash_{\text{rhs}} exp : ty}$$

$$\text{Type System}._{\text{premises}}((op\ \widetilde{args})) \equiv ps'$$
$$\text{if } r = \frac{ps}{\Gamma \vdash (op\ \widetilde{args'})} \in \text{Type System}$$
$$\text{and } \frac{ps'}{\Gamma \vdash (op\ \widetilde{args})} = r \text{ instantiated with } (op\ \widetilde{args})$$

References

Jonathan Aldrich, Robert J. Simmons, and Key Shin. 2008. SASyLF: an educational proof assistant for language theory. In *FDPE '08: Proceedings of the 2008 international workshop on Functional and declarative programming in education* (Victoria, BC, Canada). ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/1411260.1411266

David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. 2014. Abella: A System for Reasoning about Relational Specifications. *Journal of Formalized Reasoning* 7, 2 (2014). https://doi.org/10.6092/issn.1972-5787/4650

Lorenzo Bettini. 2011. A DSL for Writing Type Systems for Xtext Languages. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java* (Kongens Lyngby, Denmark) *(PPPJ '11)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/2093157.2093163

Luca Cardelli. 2004. Type Systems. In *Allen B. Tucker, Jr. (Editor-in-Chief), The Computer Science Handbook*. CRC Press, in cooperation with ACM.

James Cheney. 2005. Toward a General Theory of Names: Binding and Scope. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding* (Tallinn, Estonia) *(MERLIN 05)*. Association for Computing Machinery, New York, NY, USA, 33–40. https://doi.org/10.1145/1088454.1088459

A. Church. 1940. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic* 5 (1940), 56–68.

Matteo Cimini. 2015. *TypeSoundnessCertifier*. https://github.com/mcimini/TypeSoundnessCertifier.

Matteo Cimini. 2019. Early Experience in Teaching the Basics of Functional Language Design with a Language Type Checker. In *Trends in Functional Programming - 20th International Symposium, TFP 2019, Vancouver, BC, Canada, June 12-14, 2019, Revised Selected Papers*. 21–37. https://doi.org/10.1007/978-3-030-47147-7_2

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à La Carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. ACM, New York, NY, USA, 207–218. https://doi.org/10.1145/2429069.2429094

Sven Efftinge and Miro Spönemann. [n.d.]. *Xtext Reference Documentation*. https://eclipse.org/Xtext/documentation

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: library-based syntactic language extensibility. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (Portland, Oregon, USA) *(OOPSLA '11)*. ACM, New York, NY, USA, 391–406. https://doi.org/10.1145/2048066.2048099

Sebastian Erdweg, Tijs Storm, Markus Völter, Meinte Boersma, Remi Bosman, WilliamR. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, PedroJ. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin Vlist, Guido H. Wachsmuth, and Jimi Woning. 2013. The State of the Art in Language Workbenches. In *Software Language Engineering*, Martin Erwig, Richard F. Paige, and Eric Wyk (Eds.). Lecture Notes in Computer Science, Vol. 8225. 197–217.

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and Comparing Language Workbenches. *Comput. Lang. Syst. Struct.* 44, PA (Dec. 2015), 24–47.

Matthias Felleisen, Robert B. Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press.

Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages? https://doi.org/articles/languageWorkbench.html

Jean-Yves Girard. 1972. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. Dissertation. Université Paris VII.

Sylvia Grewe. 2019. *Automating Type Soundness Proofs for Domain-Specific Languages*. Ph.D. Dissertation. Darmstadt University of Technology, Germany. http://tuprints.ulb.tu-darmstadt.de/9025/

Sylvia Grewe, Sebastian Erdweg, and Mira Mezini. 2016. Using Vampire in Soundness Proofs of Type Systems. In *Proceedings of the 1st and 2nd Vampire Workshops (EPiC Series in Computing, Vol. 38)*, Laura Kov\'acs and Andrei Voronkov (Eds.). EasyChair, 33–51. https://doi.org/10.29007/22x6

Sylvia Grewe, Sebastian Erdweg, and Mira Mezini. 2017. Automating Proof Steps of Progress Proofs: Comparing Vampire and Dafny. In *Vampire 2016. Proceedings of the 3rd Vampire Workshop (EPiC Series in Computing, Vol. 44)*, Laura Kovacs and Andrei Voronkov (Eds.). EasyChair, 33–45. https://doi.org/10.29007/5zjp

Sylvia Grewe, Sebastian Erdweg, Pascal Wittmann, and Mira Mezini. 2015. Type Systems for the Masses: Deriving Soundness Proofs and Efficient Checkers. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (Pittsburgh, PA, USA) *(Onward! 2015)*. ACM, New York, NY, USA, 137–150. https://doi.org/10.1145/2814228.2814239

Robert Harper. 2012. *Practical Foundations for Programming Languages*. Cambridge University Press.

Robert Harper and Chris Stone. 2000. A Type-Theoretic Interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honor of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte (Eds.). MIT Press.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450. https://doi.org/10.1145/503502.503505

Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. ACM, New York, NY, USA, 444–463. https://doi.org/10.1145/1869459.1869497

Florian Lorenzen and Sebastian Erdweg. 2016. Sound type-dependent syntactic language extension. In *POPL*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 204–216. http://dl.acm.org/citation.cfm?id=2837614

Daniel Marino and Todd Millstein. 2009. A Generic Type-and-Effect System. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation* (Savannah, GA, USA) *(TLDI '09)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/1481861.1481868

Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: Reusable Engineering of Real-world Semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. ACM, New York, NY, USA, 175–188. https://doi.org/10.1145/2628136.2628143

Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Springer-Verlag, Berlin, Heidelberg, 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In *Proceedings of the 16th International Conference on Automated Deduction: Automated Deduction (CADE-16)*. Springer-Verlag, London, UK, UK, 202–206. http://dl.acm.org/citation.cfm?id=648235.753634

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge MA. 153–170 pages.

Casper Bach Poulsen, Pierre Néron, Andrew Tolmach, and Eelco Visser. 2016. Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 20:1–20:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.20

Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-typed definitional interpreters for imperative languages. 2, POPL, Article 16 (Jan. 2018), 16:1–16:?? pages. https://doi.org/10.1145/3158104

John Reynolds. 1974. Towards a Theory of Type Structure. In *Proc. Colloque sur la Programmation*. Springer-Verlag LNCS 19, New York, 408–425.

Michael Roberson, Melanie Harries, Paul T. Darga, and Chandrasekhar Boyapati. 2008. Efficient Software Model Checking of Soundness of Type Systems. *SIGPLAN Not.* 43, 10 (Oct. 2008), 493–504. https://doi.org/10.1145/1449955.1449803

Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*. 284–298. https://doi.org/10.1145/3372885.3373818

Carsten Schürmann. 2000. *Automating the Meta Theory of Deductive Systems*. Ph.D. Dissertation. Department of Computer Science, Carnegie Mellon University. http://reports-archive.adm.cs.cmu.edu/anon/2000/abstracts/00-146.html Available as Technical Report CMU-CS-00-146.

Christopher Schwaab and Jeremy G. Siek. 2013. Modular Type-safety Proofs in Agda. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification* (Rome, Italy) *(PLPV '13)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/2428116.2428120

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2007. Ott: Effective Tool Support for the Working Semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1291151.1291155

Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*. 81–92.

Eelco Visser, Guido Wachsmuth, Andrew Tolmach, Pierre Neron, Vlad Vergu, Augusto Passalaqua, and Gabrieël Konat. 2014. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Portland, Oregon, USA) *(Onward! 2014)*. Association for Computing Machinery, New York, NY, USA, 95–111.

https://doi.org/10.1145/2661136.2661149

Markus Voelter and Konstantin Solomatov. 2010. Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS. In *Third International Conference on Software Language Engineering (SLE 2010) (Lecture Notes in Computer Science)*, Mark van den Brand, Brian Malloy, and Steffen Staab (Eds.). Springer.

Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and Computation* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093